Interval Methods for Ray Casting Implicit Surfaces with Affine Arithmetic

AFFONSO DE CUSATIS JUNIOR[1] LUIZ HENRIQUE DE FIGUEIREDO[2] MARCELO GATTASS[1]

[1]Department of Computer Science, PUC-Rio, Rua Marquês de São Vicente 225, 22453-900 Rio de Janeiro, RJ, Brazil
affonso,gattass@inf.puc-rio.br

[2]LNCC–Laboratório Nacional de Computação Científica, Avenida Getúlio Vargas 333, 25651-070 Petrópolis, RJ, Brazil
lhf@tecgraf.puc-rio.br

Abstract. We study the performance of affine arithmetic as a replacement for interval arithmetic in interval methods for ray casting implicit surfaces. Affine arithmetic is a variant of interval arithmetic designed to handle the dependency problem, and which has improved several interval algorithms in computer graphics.

Keywords: image synthesis; root location; interval arithmetic; range analysis; self-validated computing.

1 Introduction

Given a function $h\colon \mathbf{R}^3 \to \mathbf{R}$, the set

$$S = \{p \in \mathbf{R}^3 : h(p) = 0\}$$

is in general a surface, which is called the *implicit surface* defined by h. In other words, an implicit surface is the set of all solutions $p = (x, y, z)$ of a single equation in three variables: $h(x, y, z) = 0$. Implicit surfaces are important primitives in CSG modeling systems [2].

An important task in the study of implicit surfaces is *rendering*, that is, the computation of an image of the set S as seen from a given point of view in $\mathbf{R}^3$. One natural technique for rendering implicit surfaces is *ray casting*: rays are cast from the viewpoint, through an image plane, until they first hit the surface S; at the intersection point, the normal to S is computed and a color is selected for the corresponding pixel in the image according to some illumination model. Ray casting is a basic step in *ray tracing*, a more sophisticated technique that can handle reflections and refractions [9].

A ray from the viewpoint E (where the *eye* is located) in the direction v is parametrized by $r(t) = E + t \cdot v$, where $t \in [0, \infty)$. Hence, the points where this ray intersects the surface S are found by solving the equation

$$f(t) = h(r(t)) = 0.$$

We are interested only in the first intersection, that is, in the *smallest* zero of f in $[0, \infty)$. Thus, to find the color of each pixel in the image, we must solve one nonlinear equation.

If h is a polynomial, then f is also a polynomial of the same degree. Thus, when the degree of h is at most 4, we can use the classical formulas [13] for computing all the roots of $f(t) = 0$. For polynomials of higher degree, there are no closed formulas, but there are robust approximation methods for finding all zeros of f based on Descartes' rule of signs [10].

There are many classical methods for solving general nonlinear equations [19], but none of those are robust enough for ray casting. We need to find the first zero reliably. Otherwise, if a different zero is found, then the normal to the surface at the corresponding point is likely to be different, the computed color will be wrong, and so will be the final image.

Therefore, successful image synthesis by ray casting depends crucially on being able to find reliably the smallest positive root of arbitrary nonlinear equations.

Reliable solution methods can be obtained by using range analysis and *interval arithmetic* [18], as described in Sections 2 and 3. Interval methods have been used successfully in ray casting [1, 16, 23] and in several other graphics problems [8, 17, 20, 22].

Interval arithmetic [18] was originally proposed for controlling error propagation in numeric computations, but of special importance to computer graphics is its ability to probe the behavior of arbitrary functions reliably over whole intervals — this is much more powerful and robust than point sampling. However, as we shall see briefly in Section 3, interval arithmetic suffers from a overestimation problem that can negatively impact the performance of interval algorithms.

Affine arithmetic, introduced in SIBGRAPI'93 [3] and briefly described in Section 4, is a variant of interval arithmetic that is more resistant to overestimation — this has led to faster algorithms for several problems in computer graphics [6, 7, 11, 12].

In this paper, we continue this research and study the performance of affine arithmetic in interval methods for ray casting implicit surfaces. As we argue in Section 5, affine arithmetic promises to be useful in ray casting. Section 6 describes our experimental results, which are discussed in Section 7 along with our conclusions.

2 Range analysis and ray casting

As mentioned in Section 1, the key to solving arbitrary non-linear equations reliably is to use range analysis instead of point sampling.

Range analysis is the study of the behavior of real functions based on estimates for their set of values. Given a function $f\colon \Omega \subseteq \mathbf{R}^n \to \mathbf{R}$, range analysis methods provide an *inclusion function* for f, that is, a function F defined on the subsets X of Ω such that

$$F(X) \supseteq f(X) = \{f(x) : x \in X\}.$$

Thus, $F(X)$ is an estimate for the *complete* set of values taken by f on X. In particular, if $0 \notin F(X)$, then there are *no* solutions of $f(t) = 0$ in X.

Therefore, inclusion functions can be used to discard intervals that cannot contain solutions. We only need to consider intervals for which $0 \in F(X)$, because these *may* contain solutions. (Range estimates are not required to be tight, that is, $F(X)$ may be strictly larger than $f(X)$. Hence, $F(X)$ may contain 0 even if $f(X)$ does not.)

Thus, given an inclusion function F for f, the simple bisection algorithm below will find the first zero of f, within a user-specified tolerance ε.

```
interval-bisection([a, b]):
  if 0 ∈ F([a, b]) then
    c ← (a + b)/2
    if (b − a) < ε then
      return c
    else
      interval-bisection([a, c])
      interval-bisection([c, b])
```

For ray casting, we start the solution of $f(t) = 0$ by calling interval-bisection($[0, T]$), where T is some large value (obtained for instance from the far clipping plane).

As mentioned above, this algorithm does not miss any zeros of f, because it only discards intervals that cannot contain zeros. Moreover, because the left half $[a, c]$ is always tested before the right half $[c, b]$, the algorithm finds the *first* zero, if any.

If we do not stop the algorithm after finding the first zero, then it will find *all* zeros of f, in the order they occur from left to right. Finding all zeros is required for ray casting CSG models [8].

The interval-bisection algorithm is an interval version of the classical bisection algorithm [19]. However, classical bisection only converges to a zero of f when started with a *bracketing interval*, that is, $f(a)$ and $f(b)$ must have different signs. Even then, there is no guarantee that it will converge to the *first* root [4].

3 Interval arithmetic

Interval arithmetic (IA) is the classical technique for range analysis. IA was invented by Moore [18] with the explicit goal of improving the reliability of numerical computation, by automatically keeping track of rounding errors. However, its ability to probe the behavior of functions reliably over whole intervals seems to us to be its more important asset, because it is a powerful tool for solving difficult problems that cannot be solved robustly by point sampling.

IA provides robust estimates for the results of numerical computations by representing numbers as intervals and extending the basic arithmetic operations and elementary functions to intervals. For example, we have

$$\begin{aligned}
[a,b] + [c,d] &= [a+c, b+d] \\
[a,b] \times [c,d] &= [\min\{ac, ad, bc, bd\}, \max\{ac, ad, bc, bd\}] \\
[a,b]^2 &= [0, \max(a^2, b^2)] \\
\exp\,[a,b] &= [\exp(a), \exp(b)].
\end{aligned}$$

Moore [18] showed that every computable function f has a natural interval extension F, which is an inclusion function for f. Any algorithm for computing f can automatically be interpreted as an algorithm for computing F simply by composing interval formulas for the primitive operations in the same way they are composed to compute the function itself.

Interval extensions are specially elegant to implement with programming languages that support operator overloading, such as C++, but they can be easily implemented in any programming language, either manually or with the aid of a precompiler [5]. There are several IA packages available in the internet [15].

A limitation of IA is that its range estimates tend to be much wider than the exact ranges, sometimes to the point of uselessness. This over-conservatism is mainly due to the implicit assumption that operands in primitive operations are mutually independent. If this assumption is false, then not all combinations of values in the operand intervals will be attained, and the result interval computed by IA may be much wider than the exact range of the result quantity.

For a simple example of this *dependency problem*, consider the interval computation of $y = x(4 - x)$, for $x \in [1, 3]$. The IA rules given above yield

$$\begin{aligned}
x &= [1,3] \\
4 - x &= [4-3, 4-1] = [1,3] \\
y = x(4-x) &= [1,9],
\end{aligned}$$

which is 8 times larger than $[3, 4]$, the exact range of y.

More generally, if $x \in [2 - u, 2 + u]$ with $0 \le u \le 1$, then the exact range of $y = x(4 - x)$ is $[4 - u^2, 4]$, and the IA rules give $y \in [(2 - u)^2, (2 + u)^2]$. Thus, the IA result has diameter $8u$, whereas the exact range has diameter u^2.

The IA result is always more than 8 times larger than the exact range. In fact, the relative diameter is $8u/u^2 = 8/u$, which actually *increases* as u decreases.

The over-conservatism of IA is particularly severe in long computation chains, where one often observes an "error explosion": as the evaluation advances down the chain, the relative accuracy of the computed intervals decreases exponentially, and they soon become too wide to be useful, by many orders of magnitude. Unfortunately, long computation chains are not uncommon in computer graphics applications.

4 Affine arithmetic

Affine arithmetic (AA), introduced by Comba and Stolfi [3], is a technique for range analysis that was designed with the explicit goal of handling the dependency problem of IA. Like standard interval arithmetic, AA can provide guaranteed bounds for the computed results, taking into account input, truncation, and rounding errors. Unlike IA, however, AA automatically keeps track of *correlations* between computed and input quantities, and is therefore more resistant to the catastrophic loss of precision often observed in long interval computations.

In AA, a quantity x is represented as an *affine form*,

$$\hat{x} = x_0 + x_1\,\varepsilon_1 + \cdots + x_n\,\varepsilon_n,$$

which is a polynomial of degree 1 in *noise symbols* ε_i, whose values are unknown but assumed to lie in the interval $[-1, +1]$. Thus, the quantity x lies in the interval $[x_0 - r, x_0 + r]$, where $r = |x_1| + \cdots + |x_n|$.

As done in IA, the basic arithmetic operations and elementary functions can be extended to handle affine forms. Affine operations (translation, scale, addition, and subtraction) are straightforward. Non-affine operations, such as multiplication and square root, use a good affine approximation plus an error term (which creates a new noise symbol). For details of how AA (and IA) operations can be implemented, see reference [21].

The key feature of AA is that the same noise symbol may contribute to the uncertainty of two or more quantities (inputs, outputs, or intermediate results) arising in the evaluation of an expression. The sharing of a noise symbol ε_i by two affine forms $\hat{x}$ and $\hat{y}$ indicates some partial dependency between the underlying quantities x and y. The magnitude and sign of the dependency is determined by the corresponding coefficients x_i and y_i.

Consider again the example given in Section 3, but now computed with AA instead of IA:

$$\begin{aligned} \hat{x} &= 2 + u\,\varepsilon_1 \\ 4 - \hat{x} &= 2 - u\,\varepsilon_1 \\ y = \hat{x}(4-\hat{x}) &= 4 + 0\,\varepsilon_1 + u^2\,\varepsilon_2, \end{aligned}$$

which implies that $y \in [4 - u^2, 4 + u^2]$. This interval has diameter $2u^2$ — only twice as large as the exact range. So, the relative diameter now remains constant at 2. Note also that the influence of the shared symbol ε_1 happened to cancel out (to first order) in y.

The improved results given by AA come at a cost: the evaluation of a function using affine forms is more expensive than the evaluation of the same function using intervals (which is more expensive than floating-point evaluation). Nevertheless, better range estimates usually imply overall faster algorithms, because fewer range estimates have to be computed, even if each individual estimate is expensive. This phenomenon has been observed in several interval methods based on AA: enumeration of implicit surfaces [7], intersection of parametric surfaces [6], sampling [12] and ray-tracing [11] procedural shaders.

5 Affine arithmetic and ray casting

There are two main reasons for expecting AA to give good results in ray casting an implicit surface given by $h(x, y, z) = 0$.

First, AA automatically notices the linear correlations between x, y, and z when the point (x, y, z) lies on a ray, and exploits them when computing the value of $h(x, y, z)$. IA, on the other hand, sees the point (x, y, z) merely inside an axis-aligned box, and has to evaluate h on all points of this (usually large) box, even those far away from the ray. Thus, IA may conclude that the ray intersects the surface merely because its bounding box does.

Second, we can exploit the additional information provided by AA to reduce the interval where a root could be located, at almost no extra cost: While IA approximates a function f in an interval $[a, b]$ by a rectangle (Figure 1), AA approximates f by a parallelogram (Figure 2). (We shall presently see why.)

Thus, any root of f in $[a, b]$ must lie in the intersection of the parallelogram with $[a, b]$, which is sometimes much smaller than $[a, b]$ (as in Figure 2). Actually, testing whether this intersection is non-empty is equivalent to testing whether $0 \in F([a, b])$, thus simplifying the interval bisection algorithm given in Section 2.

To see that the AA approximation of f for $t \in [a, b]$ is a parallelogram, we first write $\hat{t} = t_0 + t_1\,\varepsilon_1$, where $t_0 = (b + a)/2$ and $t_1 = (b - a)/2$. Then, AA gives

$$\hat{f} = f_0 + f_1\,\varepsilon_1 + \cdots + f_n\,\varepsilon_n.$$

Note that ε_1 is the same noise symbol in both forms. The other terms come from affine approximations to the non-affine operations required to compute f. Since we want to compare t and f, these terms are not important individually; we can condense them into a single term and write

$$\hat{f} = f_0 + f_1\,\varepsilon_1 + f_k\,\varepsilon_k,$$

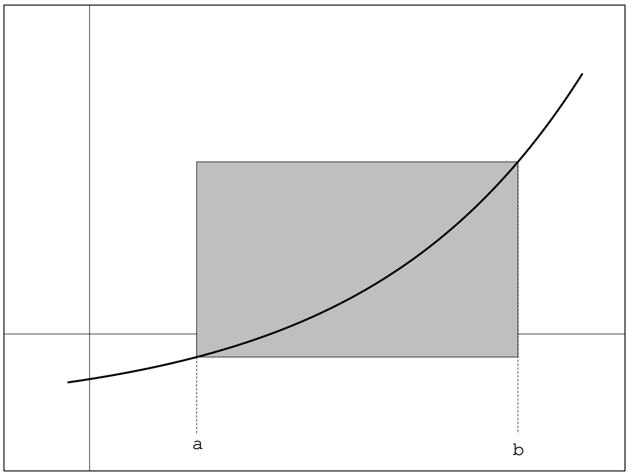


Figure 1: Geometry of IA approximation.

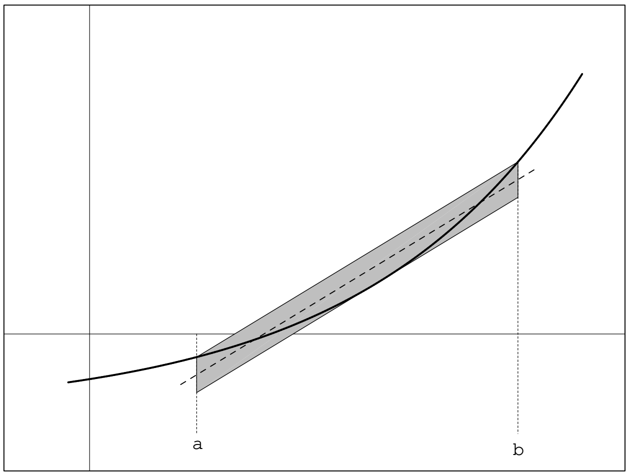


Figure 2: Geometry of AA approximation.

where $f_k = |f_2| + \cdots + |f_n|$. When ε_k is held fixed, the point (t, f) sweeps a line segment as ε_1 varies in $[-1, 1]$. Thus, as ε_k varies in $[-1, 1]$, this line segment vertically sweeps a parallelogram of the form depicted in Figure 2.

6 Results

We compared the performance of AA with the performance of IA for ray casting the implicit surfaces shown in Figure 3, whose equations are given in Table 1. Some surfaces have been tested with their expanded equations too (marked $+$ in Table 1), to see whether increasing the complexity of the expressions, thus adding correlations, would benefit AA.

Besides the pure interval method described in Section 2, we also tested a hybrid method that uses range estimates for the derivative to isolate the roots, and classical methods to refine them, as done by Mitchell [16].

The hybrid algorithm below uses an inclusion function G for the derivative f', obtained with IA or AA. If $0 \notin G([a, b])$, then f is monotonic in $[a, b]$ and can have at most one root in this interval. This root is then refined using either classical bisection or the more sophisticated Brent's method [19].

```
hybrid-bisection([a, b]):
    if 0 ∈ F([a, b]) then
        c ← (a + b)/2
        if (b − a) < ε then
            return c
        if 0 ∉ G([a, b]) then
            refine root with classical method
        else
            hybrid-bisection([a, c])
            hybrid-bisection([c, b])
```

We implemented a ray caster in C using Borland C++ Builder. Libraries for IA and AA were written having the same API, so that testing each variant was a simple matter of linking with the right library. We used Maple™ to compute optimized expressions for each function and their partial derivatives, and a small precompiler in Lua [14] to convert these expressions into calls to the arithmetic libraries.

Table 2 shows the time required to generate a 64×64 ray-casting image of each surface in Table 1, using a 166 MHz Pentium running Windows 95. (The images in Figure 3 are 200×200, but the relative times are essentially independent of image resolution.) The AA variant exploits the additional information described in Section 5, whereas the AA$'$ variant does not. A star $\star$ marks the best time; a diamond $\diamond$ means that AA$'$ was already faster than pure IA. Table 3 shows the number of interval evaluations required in each case. (This number was only recorded for the pure variants in the case of expanded equations.)

7 Conclusion

The results indicate that AA is indeed useful as a replacement for IA in interval methods for ray casting implicit surfaces, specially when the function defining the surface is complicated. Exploiting the additional information that AA provides (as described in Section 5) plays an important part in the performance of the algorithm. Even AA alone can lead to faster algorithms, when the function is very complicated (e.g., the expanded double torus and the Mitchell surface).

Table 2 suggests that hybrid methods are not faster than pure interval methods, probably because the cost of computing interval estimates for derivatives offsets the gains of refining roots using fast floating-point arithmetic.

Nevertheless, more examples may be needed to fully establish the advantages of using AA in this context. We suspect that the Borland C++ Builder environment is stealing machine cycles for its user interface, and that could be interfering with our timings. We are currently working on a different, batch implementation under Linux. Preliminary results using this implementation are very promising, and will be reported elsewhere.

AA did not give the best results in all cases. AA suffers from an overestimation problem for some primitive operations, such as the square operation ($y \leftarrow x^2$) [21]. Accordingly, IA was faster for the surfaces that contain many square terms, with no negative correlation.

Range estimates given by AA are not always better than those given by IA. For the function in Figure 2, the range estimate computed by AA is actually larger than the range estimate computed by IA. Nevertheless, AA still provides better approximations than IA in the sense that the area of the AA parallelogram is smaller than the area of the IA rectangle. This leads to better range estimates for small input intervals: AA estimates converge quadratically, whereas IA estimates converge linearly. (See the example in Section 4 and more details in reference [21].)

An important goal of the research with AA is to try to understand the cases when the use of AA is recommended. As in previous AA work [6, 7, 11, 12], we have found that AA gives best results when the functions are very complicated. Thus, a natural next step in this research is the extension of these algorithms and experiments to CSG models, specially complex blobby models, which are very popular in implicit modeling.

Finally, like Mitchell [16], we have found that rounding-error control was not necessary for generating correct images. This was good for two reasons: first, we did not have to implement rounding-error control in our IA and AA libraries, which is done in a slightly different way in each platform; second, changing the rounding mode is costly in some platforms, and this could affect our timings.

Acknowledgements. We thank J. Stolfi for supplying code and expertise about AA. The authors are partially supported by research grants from the Brazilian Council for Scientific and Technological Development (CNPq). This research is part of the first author's M.Sc. thesis at PUC-Rio.

References

[1] W. Barth, R. Lieger, and M. Schindler. Ray tracing general parametric surfaces using interval arithmetic. *The Visual Computer*, 10(7):363–371, 1994.

[2] J. Bloomenthal, editor. *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.

[3] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. *Proceedings of SIBGRAPI'93*, pages 9–18, October 1993.

[4] G. Corliss. Which root does the bisection algorithm find? *SIAM Review*, 19(2):325–327, 1977.

[5] F. D. Crary. A versatile precompiler for nonstandard arithmetics. *ACM Transactions on Mathematical Software*, 5(2):204–217, 1979.

[6] L. H. de Figueiredo. Surface intersection using affine arithmetic. In *Proceedings of Graphics Interface'96*, pages 168–175, May 1996.

[7] L. H. de Figueiredo and J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15(5):287–296, 1996.

[8] T. Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *Computer Graphics*, 26(2):131–138, July 1992. (SIGGRAPH'92 Proceedings).

[9] A. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[10] P. Hanrahan. Ray tracing algebraic surfaces. *Computer Graphics*, 17(3):83–90, July 1983. (SIGGRAPH '83 Proceedings).

[11] W. Heidrich and H.-P. Seidel. Ray-tracing procedural displacement shaders. In *Proceedings of Graphics Interface'98*, pages 8–16, June 1998.

[12] W. Heidrich, P. Slusallik, and H.-P. Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, 17(3):158–176, 1998.

[13] D. Herbison-Evans. Solving quartics and cubics for graphics. In A. Paeth, editor, *Graphics Gems V*, pages 3–15. Academic Press, 1995.

[14] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua: an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[15] V. Kreinovich. Interval software. `http://cs.utep.edu/interval-comp/intsoft.html`.

[16] D. P. Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface'90*, pages 68–74, May 1990.

[17] D. P. Mitchell. Three applications of interval analysis in computer graphics. In *Frontiers in Rendering course notes*, pages 14-1–14-13. SIGGRAPH'91, July 1991.

[18] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[19] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge University Press, 1992.

[20] J. M. Snyder. Interval analysis for computer graphics. *Computer Graphics*, 26(2):121–130, July 1992. (SIGGRAPH'92 Proceedings).

[21] J. Stolfi and L. H. de Figueiredo. *Self-Validated Numerical Methods and Applications*. Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro, 1997.

[22] K. G. Suffern and E. D. Fackerell. Interval methods in computer graphics. *Computers and Graphics*, 15(3):331–340, 1991.

[23] D. L. Toth. On ray tracing parametric surfaces. *Computer Graphics*, 19:171–179, July 1985. (SIGGRAPH '85 Proceedings).

Surface	$h(x,y,z)$
Sphere	$x^2+y^2+z^2-1$
Drop	$4(x^2+y^2)-(1+z)(1-z)^3$
Drop+	$4x^2+4y^2-1+2z-2z^3+z^4$
Torus	$(x^2+y^2+z^2-1-0.25)^2-4(x^2+y^2)$
Double torus	$(4x^2(1-x^2)-y^2)^2+z^2-0.25$
Double torus+	$16x^4-32x^6-8x^2y^2+16x^8+8x^4y^2+y^4+z^2-0.25$
Mitchell [16]	$4(x^4+(y^2+z^2)^2)+17x^2(y^2+z^2)-20(x^2+y^2+z^2)+17$
Six-peak	$(3x^2-y^2)^2y^2-(x^2+y^2)^4-z$
Steiner	$x^2y^2+y^2z^2+z^2x^2+xyz$

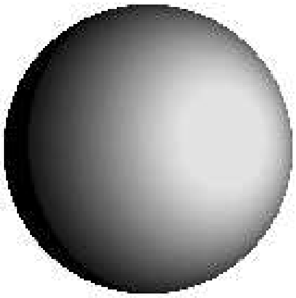
Table 1: The equations of the implicit surfaces used in the experiments.

	IA			AA			AA′		
	pure	bisection	Brent	pure	bisection	Brent	pure	bisection	Brent
Sphere	115	98	98	77 ⋆	83	82	154	116	116
Drop	143 ⋆	150	148	226	304	305	297	340	340
Drop+	269	—	—	165 ⋆	—	—	275	—	—
Torus	390	247	246	115 ⋆	203	203	302	387	385
Double torus	192 ⋆	193	193	300	574	572	478	1060	1050
Double torus+	2885	—	—	730 ⋆	—	—	1130 ⋄	—	—
Mitchell	1164	550	542	285 ⋆	519	520	642 ⋄	1134	1130
Six-peak	340	395	396	260 ⋆	705	703	560	1270	1260
Steiner	330 ⋆	422	422	460	1052	1040	554	1376	1380

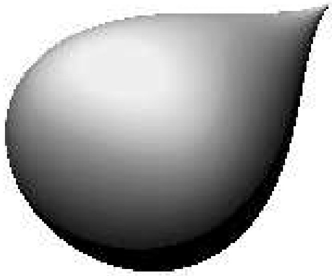
Table 2: Time required to ray cast the example surfaces (in centiseconds).

	IA		AA		AA′	
	pure	hybrid	pure	hybrid	pure	hybrid
Sphere	95	42	28	16	82	33
Drop	89	54	74	68	110	77
Drop+	130	—	48	—	97	—
Torus	317	89	43	41	148	75
Double torus	123	61	109	104	210	165
Double torus+	57	—	153	—	244	—
Mitchell	470	157	71	57	187	148
Six-peak	242	131	87	81	203	158
Steiner	176	131	124	118	167	160

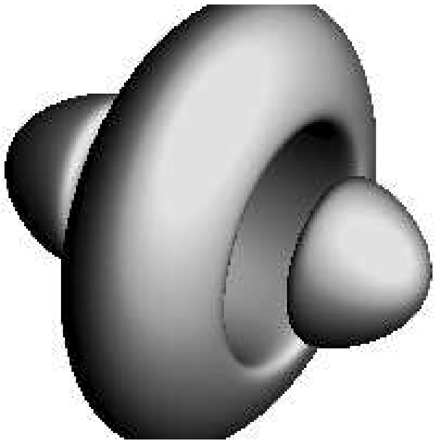
Table 3: Number of interval evaluations (in thousands).



(a) Sphere

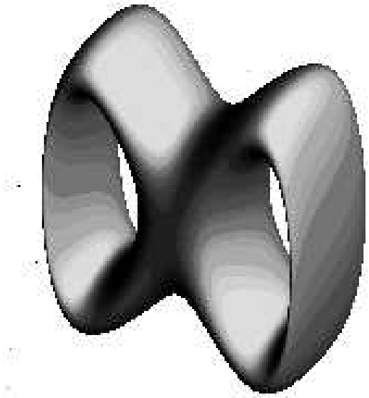


(b) Drop

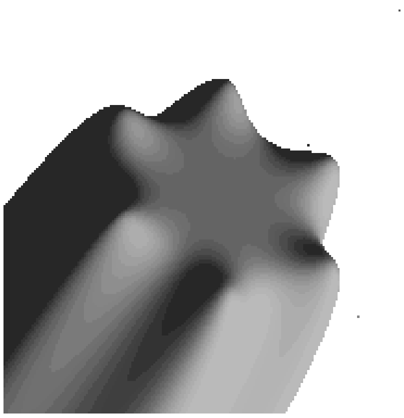


(c) Mitchell

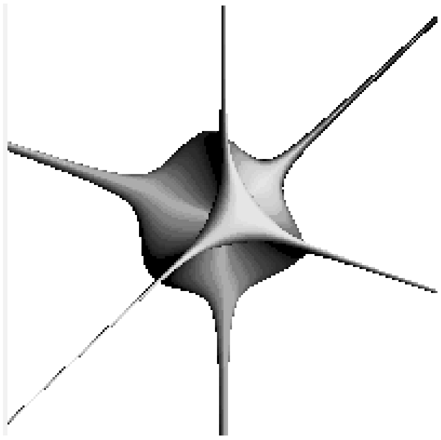
(d) Torus



(e) Double Torus



(f) Six-peak



(g) Steiner

Figure 3: The surfaces used in the experiments.

These images are available in 24-bit color at `http://www.tecgraf.puc-rio.br/~lhf/sib99/`.